TUGAS AKHIR – TJ141502

IMPLEMENTASI *RAFT CONSENSUS ALGORITHM* SEBAGAI REPLIKASI *STORAGE SERVER* MENGGUNAKAN *USER DATAGRAM PROTOCOL*

ARIF NUR KHOIRUDIN
NRP 2913 100 032

Dosen Pembimbing
Mochamad Hariadi, ST., M.Sc., Ph.D.
Dr. Supeno Mardi Susiki Nugroho, ST., MT.

DEPARTEMEN TEKNIK KOMPUTER
Fakultas Teknologi Elektro
Institut Teknologi Sepuluh Nopember
Surabaya 2017

[Halaman ini sengaja dikosongkan].

FINAL PROJECT – TJ141502

Implementation of Raft Consensus Algorithm for Storage Server Replication Using User Datagram Protocol

ARIF NUR KHOIRUDIN
NRP 2913 100 032

Advisor
Mochamad Hariadi, ST., M.Sc., Ph.D.
Dr. Supeno Mardi Susiki Nugroho, ST., MT.

Departement of Computer Engineering
Faculty of Electrical Technology
Sepuluh Nopember Institute of Technology
Surabaya 2017

[Halaman ini sengaja dikosongkan].

PERNYATAAN KEASLIAN TUGAS AKHIR

Dengan ini saya menyatakan bahwa isi sebagian maupun keseluruhan Tugas Akhir saya dengan judul "**Implementasi *Raft Consensus Algorithm* Sebagai Replikasi *Storage Server* menggunakan *User Datagram Protokol***" adalah benar-benar hasil karya intelektual mandiri, diselesaikan tanpa menggunakan bahan-bahan yang tidak diijinkan dan bukan karya pihak lain yang saya akui sebagai karya sendiri.

Semua referensi yang dikutip maupun dirujuk telah ditulis secara lengkap pada daftar pustaka.

Apabila ternyata pernyataan ini tidak benar, saya bersedia menerima sanksi sesuai peraturan yang berlaku.

Surabaya, Juni 2017

Arif Nur Khoirudin

NRP. 2913100032

LEMBAR PENGESAHAN

Implementasi *Raft Consensus Algorithm* Sebagai Replikasi *Storage Server* Menggunakan *User Datagram Protocol*

Tugas Akhir ini disusun untuk memenuhi salah satu syarat memperoleh gelar Sarjana Teknik di Institut Teknologi Sepuluh Nopember Surabaya

Oleh : Arif Nur Khoirudin (NRP: 2913100032)

Tanggal Ujian : 12 Juli 2017 **Periode Wisuda : September 2017**

Disetujui oleh:

(Pembimbing I)

Mochamad Hariadi, ST., M.Sc., Ph.D.
NIP: 196912091997031002

(Pembimbing II)

Dr. Supeno Mardi Susiki N, ST., M.T.
NIP: 197003131995121001

(Penguji I)

Prof. Dr. Ir. Mauridhi Hery P, M.Eng.
NIP: 195809161986011001

(Penguji II)

Dr. Adhi Dharma Wibawa, ST., MT.
NIP: 197605052008121003

(Penguji III)

Muhtadin, ST., MT.
NIP: 198106092009121003

Mengetahui
Kepala Departemen Teknik Komputer

Dr. I Ketut Eddy Purnama, S.T., M.T.
NIP. 196907301995121001

ABSTRAK

Nama Mahasiswa	:	Arif Nur Khoirudin
Judul Tugas Akhir	:	Implementasi *Raft Consensus Algorithm* Sebagai Replikasi *Storage Server* Menggunakan *User Datagram Protocol*
Pembimbing	:	1. Mochamad Hariadi, ST., M.Sc., Ph.D. 2. Dr. Supeno Mardi Susiki Nugroho, ST., MT.

Replikasi sangat penting dilakukan untuk menanggulangi suatu masalah seperti *server down* atau koneksi terputus, replikasi juga dapat dilakukan untuk membagi beban *server* (*load balancing*) pada saat mengakses data. Oleh karena itu, untuk data yang akurat dan selalu tersedia maka replikasi adalah hal yang wajib untuk diimplementasikan pada infrastruktur *storage server*. Berbeda dengan *TCP*, dengan *UDP* proses replikasi menjadi lebih cepat dan ringan. Namun tantangan yang dihadapi adalah data *loss* karena *UDP* adalah *connectionless* protokol sehingga membuatnya menjadi *reliable* adalah tantangan tersendiri. Pada penelitian ini akan dilakukan sebuah implementasi replikasi *storage server* berdasar algoritma konsensus raft dengan *UDP*. Raft adalah algoritma konsensus yang mudah dimengerti dan sepenuhnya dapat di implemantasikan, pada dasarnya Raft sama dengan Paxos dalam segi *fault-tolerance* dan performa namun strukturnya berbeda dengan paxos[1]. Untuk *storage server* akan menggunakan *leveldb*, data yang disimpan adalah sebuah *chunk* dari file. Proses implementasi dilakukan dengan menerapkan paradigma *event-driven programming* dan bersifat *asynchronous non-blocking*. Telah dilakukan beberapa pengujian dan dapat disimpulkan bahwa sistem ini dapat dengan baik melakukan replikasi data.

Kata Kunci : *Cloud Computing*, *Distributed System*, *Networking*

Halaman ini sengaja dikosongkan

ABSTRACT

Name	:	Arif Nur Khoirudin
Title	:	*Implementation of Raft Consensus Algorithm for Storage Server Replication Using User Datagram Protocol*
Advisors	:	1. Mochamad Hariadi, ST., M.Sc., Ph.D. 2. Dr. Supeno Mardi Susiki Nugroho, ST., MT.

Replication is very important to be done to overcome a problem such as server down or connection problem, replication can also be done to divide the server load (load balancing) when accessing data. Therefore, for data accuracy and availability then replication is mandatory to be implemented on the storage servers. In contrast to TCP, with UDP the replication process becomes faster and lighter. But the challenge faced is data loss because UDP is a connectionless protocol that can make it reliable is a good challenge. In this research will be done a replication of storage server based on Raft Consensus Algorithm with UDP. Raft is a consensus algorithm that is easy to understand and fully implemented, basically Raft is the same as Paxos in terms of fault-tolerance and performance but the structure is different. For server storage will use leveldb, the stored data is a chunk of the file. Implementation is done by applying an event-driven programming paradigm and it is asynchronous non-blocking. There have been several tests and it can be concluded that this system can properly perform data replication.

Keywords : Cloud Computing, Distributed System, Networking

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji dan syukur kehadirat Allah SWT atas segala limpahan berkah, rahmat, serta hidayah-Nya, penulis dapat menyelesaikan penelitian ini dengan judul **Implementasi Raft Consensus Algorithm Sebagai Replikasi Storage Server Menggunakan User Datagram Protocol**.

Penelitian ini disusun dalam rangka pemenuhan bidang riset di Jurusan Teknik Komputer ITS, bidang studi telematika, serta digunakan sebagai persyaratan menyelesaikan pendidikan S1. Penelitian ini dapat terselesaikan tidak lepas dari bantuan berbagai pihak. Oleh karena itu, penulis mengucapkan terima kasih kepada:

1. Keluarga, Ibu, Bapak dan Saudara tercinta yang telah memberikan dorongan spiritual dan material dalam penyelesaian buku penelitian ini.
2. Bapak Mochamad Hariadi, ST., M.Sc., Ph.D. serta Dr. Supeno Mardi Susiki Nugroho, ST.,MT. selaku dosen pembimbing yang senantiasa mengarahkan serta memberi koreksi pada setiap kesalahan pengerjaan.
3. Bapak Jagad Hariseno, saudara Alm. Muhammad Ikhsan, serta seluruh keluarga besar Catfiz Messenger yang selalu memberikan bimbingan teknis serta memberikan banyak masukan.
4. Bapak/Ibu dosen, karyawan, serta seluruh keluarga besar Teknik Komputer ITS.
5. Seluruh teman-teman *B201-crew* Laboratorium Teknik Komputer dan Telematika ITS.

Kesempurnaan hanya milik Allah SWT, untuk itu penulis memohon segenap kritik dan saran yang membangun. Semoga penelitian ini dapat memberikan manfaat bagi kita semua. Amin.

Surabaya, Juni 2017

Penulis

Halaman ini sengaja dikosongkan

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

Halaman ini sengaja dikosongkan

DAFTAR KODE

Halaman ini sengaja dikosongkan

BAB 1
PENDAHULUAN

Penelitian ini di latar belakangi oleh berbagai kondisi yang menjadi acuan. Selain itu juga terdapat beberapa permasalahan yang akan dijawab sebagai luaran dari penelitian.

1.1 Latar belakang

Setiap hari trilyunan *bytes* data dikirimkan melalui internet. Data ini datang dari mana saja: sosial media, berita, foto, video, transaksi perbankan, dan lain sebagainya. Sesuai dengan karakteristiknya, data harus akurat, *valid*, *reliable*, relevan dan sepenuhnya tidak berkurang[3].

Dengan kondisi lalu lintas data seperti saat ini, teknologi penyimpanan data juga harus memenuhi kebutuhan. Sistem teknologi basis data saat ini telah mencapai tingkat yang lebih tinggi, dari *SQL database* hingga *No-SQL database*[4]. Perkembangan juga terjadi pada teknologi distribusi data seperti replikasi dan *clustering*[5].

Demi memenuhi karakteristik data, maka transaksi data biasanya menggunakan *TCP (Transmission Control Protocol)* dimana dilakukan inisiasi koneksi terlebih dahulu sehingga *TCP* disebut juga dengan *connection oriented protocol*. Seluruh data akan dikirim dengan sedemikian rupa sehingga paket yang *loss* akan dikirim kembali hingga keseluruhan data yang diterima akan benar-benar utuh[6]. Begitu juga pada koneksi yang digunakan untuk replikasi.

Replikasi adalah suatu hal yang wajib dilakukan untuk menyediakan layanan *storage server* yang stabil dan akurat. dengan replikasi kesalahan-kesalahan pada sistem dapat ditoleransi karena data tidak hanya disimpan di satu tempat saja.

Algoritma konsensus raft membuat sistem yang *reliable* untuk sebuah data replikasi dan terdistribusi[7], namun hampir seluruh implementasi algoritma ini menggunakan *TCP*. Jika kita melihat transaksi data pada saat ini maka penggunaan *TCP* akan membuat beban di *instance server, router*, dan komponen jaringan replikasi

akan semakin tinggi dan cenderung lebih lambat karena kompleksitas dari *TCP* itu sendiri. Di sisi lain protokol *UDP* adalah protokol yang sangat ringan, *UDP* biasa digunakan oleh aplikasi yang lebih mengutamakan kecepatan daripada jaminan data akan tersampaikan. *UDP* juga dapat digunakan sebagai transfer data seperti yang telah dilakukan oleh protokol *media sharing torrent* dan beberapa media streaming.

Jika Algoritma konsensus raft telah menyajikan sebuah sistem distribusi data yang *reliable*, maka protokol jaringan tidak perlu menggunakan protokol yang *reliable* dan cenderung berat karena *realibility* sistem sudah dijamin di *layer* atasnya yaitu dengan algoritma konsensus raft.

1.2 Permasalahan

Selama ini *UDP* tidak digunakan sebagai protokol replikasi *storage* karena dianggap tidak *reliable* sehingga data tidak tereplikasi dengan baik, namun penggunaan *TCP* yang cenderung rumit akan menyebabkan beban tinggi pada infrastruktur dan cenderung lebih lambat mengingat kebutuhan akan transaksi data kecepatan tinggi [6].

Masalah yang dirumuskan dalam penelitian ini adalah :

1. UDP tidak memiliki jaminan atau konfirmasi apakah data yang dikirim telah diterima
2. Ukuran paket data UDP (*datagram*) sangat kecil sehingga rentan terhadap loss paket apabila mengirimkan data yang besar

1.3 Tujuan

Adapun tujuan dari penelitiann ini adalah bertambahnya sebuah alternatif replikasi data untuk *storage server* yang *reliable* menggunakan algoritma konsensus raft dan UDP.

1.4 Batasan masalah

Batasan masalah dari pengerjaan tugas akhir ini adalah:

1. *Storage Engine* yang digunakan dalam penelitian ini adalah *Leveldb* karena menggunakan *key-value model* sehingga akan

lebih mudah digunakan untuk menyimpan data pada penelitian ini.

2. *Operating System* yang digunakan adalah *GNU/Linux* distribusi berbasis *Debian* dengan minimal *kernel* versi 2.6.

3. Penelitian ini hanya meneliti terkait akurasi data, performa dan keamanan masih dihiraukan.

4. Sistem ini di desain untuk replikasi yang berada di jaringan lokal saja dimana tingkat data *loss* cenderung lebih rendah.

5. Akan digunakan 4 buah *host/node* sebagai percobaan dimana 1 *host* sebagai *front-end* dan 3 *host* sebagai *storage server*.

6. Data yang digunakan sebagai percobaan adalah *binary* data dengan ukuran maksimal 1GB.

1.5 Sistematika Penulisan

Laporan penelitian Tugas akhir ini tersusun dalam sistematika dan terstruktur sehingga mudah dipahami dan dipelajari oleh pembaca maupun seseorang yang ingin melanjutkan penelitian ini. Alur sistematika penulisan laporan penelitian ini yaitu :

1. BAB I Pendahuluan
   Bab ini berisi uraian tentang latar belakang permasalahan, penegasan dan alasan pemilihan judul, sistematika laporan, tujuan dan metodologi penelitian.

2. BAB II Dasar Teori/Daftar Pustaka
   Pada bab ini berisi tentang uraian secara sistematis teori-teori yang berhubungan dengan permasalahan yang dibahas pada penelitian ini. Teori-teori ini digunakan sebagai dasar dalam penelitian, yaitu informasi terkait algoritma konsensus raft, replikasi data, dan teori-teori penunjang lainya.

3. BAB III Perancangan Sistem dan Impementasi
   Bab ini berisi tentang penjelasan-penjelasan terkait sistem yang akan dibuat. Guna mendukung itu digunakanlah blok diagram atau *work flow* agar sistem yang akan dibuat dapat terlihat dan mudah dibaca untuk implentasi pada pelaksanaan tugas akhir.

4. BAB IV Pengujian dan Analisa
   Bab ini menjelaskan tentang pengujian yang dilakukan terhadap sistem dalam penelitian ini dan menganalisa sistem. Spesifikasi perangkat keras dan perangkat lunak yang diuji juga disebutkan dalam bab ini.
5. BAB V Penutup
   Bab ini merupakan penutup yang berisi kesimpulan yang diambil dari penelitian dan pengujian yang telah dilakukan. Saran dan kritik yang membangun untuk pengembangkan lebih lanjut juga dituliskan pada bab ini.

1.6 Relevansi

Penelitian mengenai *distributed system* merupakan bidang yang sangat dibutuhkan dan dipakai dalam pemenuhan kebutuhan industri teknologi informasi yang semakin berkembang. Dengan berkembangnya layanan teknologi pada saat ini, maka kebutuhan akan teknologi penyimpanan data yang cepat, akurat, *reliable* dan *scalable* sangat dibutuhkan. Sistem replikasi dengan menggunakan algoritma konsensus raft serta protokol *UDP* ini dapat menjadi salah satu solusi akan kebutuhan sistem penyimpanan data pada saat ini.

BAB 2
TINJAUAN PUSTAKA

Demi mendukung penelitian ini, dibutuhkan beberapa teori penunjang sebagai bahan acuan dan referensi. Dengan demikian penelitian ini menjadi lebih terarah.

2.1 Algoritma Konsensus Raft

Untuk mempermudah pengertiannya maka dapat dibagi menjadi dua buah topik yang berbeda yaitu algoritma konsensus serta raft.

2.1.1 Algoritma Konsensus

Problem utama dari *distributed computing* adalah menjaga *realibility* dari kesuluruhan sistem dan mengurangi kesalahan proses. Untuk melakukan hal itu maka keseluruhan *agent/node* harus melakukan suatu persetujuan tertentu pada saat sistem berjalan. Contoh dari penggunaan konsensus adalah persetujuan atas identitas *leader*, *state machine replication*, dan *atomic broadcast*[8].

Dengan algoritma konsensus maka beberapa *node* tetap akan dapat berjalan dengan baik jika salah satu atau beberapa *node* mengalami kerusakan sistem, oleh karena itu penentuan identitas sebuah node menjadi sangat penting pada algoritma konsensus.

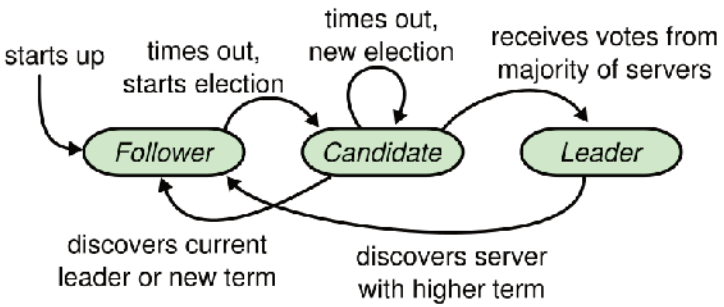
Beberapa protokol konsensus yaitu *Phase King Algorithm*, *Paxos Consensus Algorithm*, *Lockstep Protocol*, *Raft Consensus Algorithm*.

2.1.2 Raft

Raft adalah sebuah protokol algoritma konsensus yang mudah dimengerti[1]. pada dasarnya raft sama dengan paxos[7] dalam segi *fault-tolerance* dan performa, namun strukturnya berbeda dengan paxos. hal ini membuat Raft lebih mudah dimengerti dan memudahkan untuk membangun implementasinya.

Secara garis besar raft dibagi menjadi beberapa bagian, bagian yang paling mendasar yaitu *Leader Election* dan *Log Replication*

Leader Election

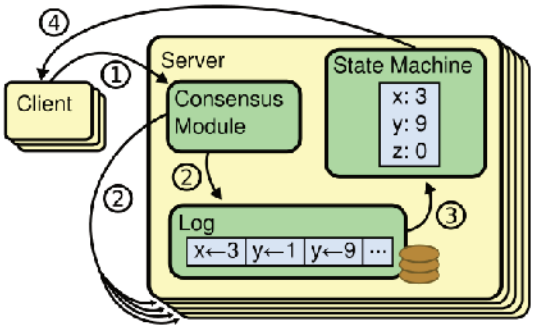


Gambar 2.1: Proses penentuan *leader* dan *follower*[1]

Gambar 2.1. menunjukkan proses perubahan status *node* dari awal mula berjalan hingga penentuan *leader* serta *follower*. Pada posisi awal, semua *node* mempunyai status sebagai *follower* dengan nilai variabel *timeout* masing-masing, ketika mencapai *timeout* maka status *follower* menjadi kandidat dan mengoleksi *vote* dari semua *node*, jika ada beberapa *node* yang mempunyai jumlah *vote* yang sama maka proses sebelumnya diulangi kembali dengan *random* timeout yang berbeda hingga terpilih satu *node* yang mempunyai *vote* tertinggi. Setelah itu *node* dengan vote tertinggi mengganti statusnya menjadi *leader* dan node lainnya sebagai *follower*[1][9].

Log Replication

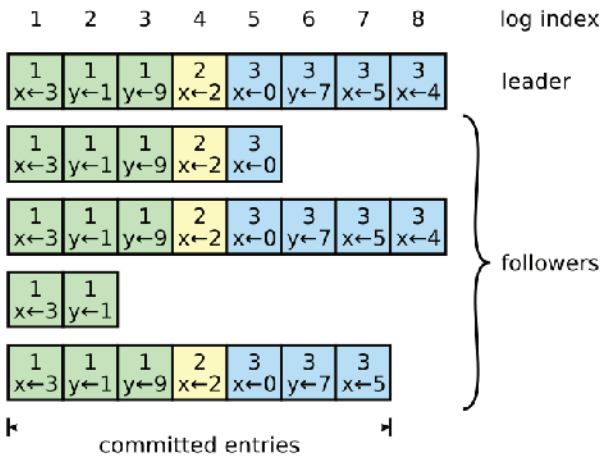
Raft melakukan manajemen replikasi data dengan cara melakukan replikasi *log* yang disebut *replicated state machine* dimana *log* inilah yang digunakan sebagai penentuan *state* pada tiap *node*.



Gambar 2.2: Replicated state machine[1]

Replicated state machine biasanya diimplementasikan dengan *log* yang tereplikasi seperti pada Gambar 2.2. Setiap *server* menyimpan sebuah *log* yang terdapat beberapa perintah dari *client*, setiap *log* mempunyai perintah yang sama dan urutan yang sama dimana selanjutnya akan dieksekusi oleh *server* tersebut dengan berurutan kamudian akan mengeluarkan hasil kembalian yang sama pada setiap *sequence* perintahnya.

Dalam sistem ini *leader* akan selalu mempunyai data terlengkap dan mempunyai *index log* paling tinggi sebelum semua data diterima oleh *follower*. Dapat dilihat pada Gambar 2.3, pada *sequence* waktu yang sama data yang disimpan di seluruh node *server* belum tentu sama, namun *leader* pasti mempunyai data yang paling lengkap dan *follower* akan terus menerima data dari *leader* hingga mempunyai *index* yang sama [1][9].



Gambar 2.3: Replicated state machine[1]

2.2 UDP (User Datagram Protocol)

Sama halnya dengan *TCP*, *UDP (User Datagram Protocol)* adalah protokol yang sering digunakan di internet. Namun *UDP* tidak pernah digunakan untuk mengirim data penting seperti halaman *web*, informasi *database*, *storage*, dan lain-lain.*UDP* biasa digunakan untuk *streaming* radio, *RTC*, dan data lain yang membutuhkan kecepatan tinggi. Alasan mengapa *UDP* lebih cepat adalah karena *UDP* tidak mempunyai *flow control* atau *error correction* [6].

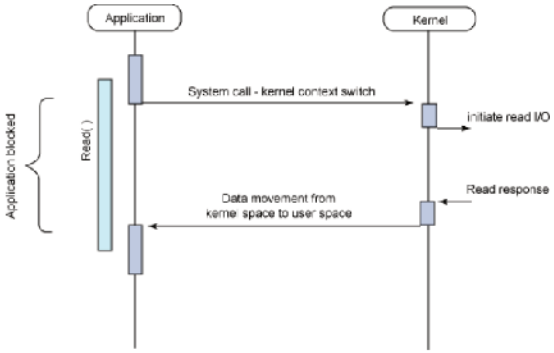
UDP adalah layer *transport protocol* yang tidak memberikan garansi suatu *realibility* dan apakah paket benar-benar sampai ke tujuan. *UDP* mendukung *multicasting* serta *broadcasting*, *UDP* sangat mendukung aplikasi yang memilih paket *loss* daripada paket sampai dalam waktu yang lama. Paket *UDP* disebut juga *datagram* dimana dibagi menjadi *header* dan *payload*[6].

2.3 Asynchronous Non Blocking I/O

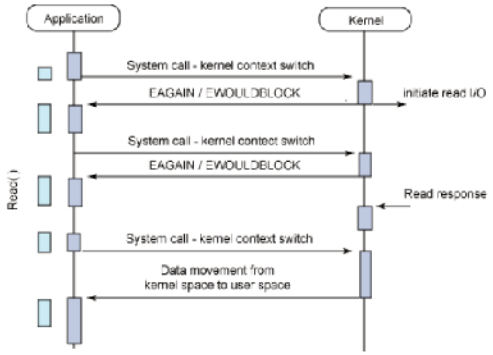
Asynchronous I/O mulai masuk pada fitur standar *linux kernel* versi 2.6. *Asyncronous non-blocking* digunakan oleh sebuah proses

dimana proses tersebut diperbolekan melakukan operasi I/O tanpa melakukan block atau menunggu proses sebelumnya selesai.

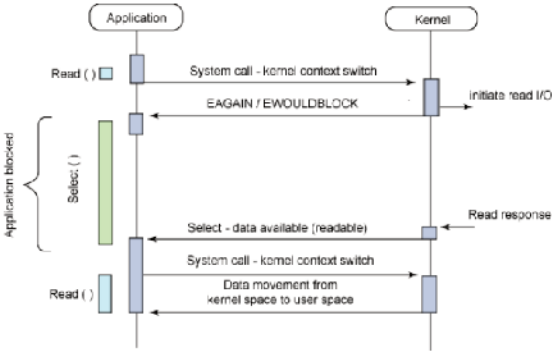
Pada dasarnya *linux I/O* mempunyai beberapa kombinasi model yaitu *syncronous blocking*, *synchronous non-blocking*, *asynchronous blocking*, dan *asynchronous non-blocking*.



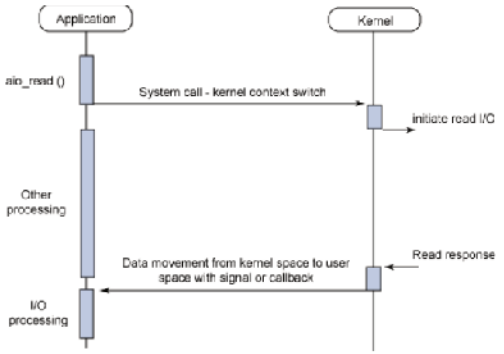
Gambar 2.4: Synchronous blocking I/O[2]



Gambar 2.5: Synchronous non-blocking I/O[2]



Gambar 2.6: Asynchronous blocking I/O[2]



Gambar 2.7: Asynchronous non-blocking I/O[2]

Dari beberapa gambar beberapa *I/O* model diatas (Gambar 2.4 - 2.7)maka dapat dilihat model yang mempunyai performa paling baik yaitu *asynchronous non-blocking I/O*. model *blocking* membutuhkan aplikasi untuk untuk melakukan *blocking* saat melakukan *I/O*, artinya proses selanjutnya arus menunggu dan tidak melakukan proses di waktu yang sama. *synchronous non-blocking* memperbolehkan proses di waktu yang sama *(overlap processing)* namun harus melihat status dari operasi I/O untuk disimpan secara terus menerus[2].

BAB 3
DESAIN DAN IMPLEMENTASI SISTEM

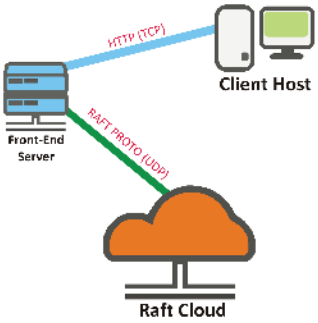
Penelitian ini dilaksanakan sesuai dengan desain sistem berikut dengan implementasinya. Desain sistem merupakan konsep dari pembuatan dan perancangan infrastruktur dan kemudian diwujudkan dalam bentuk blok-blok alur yang harus dikerjakan. Pada bagian implementasi merupakan pelaksanaan teknis untuk setiap blok pada desain sistem.

3.1 Desain Sistem

Sistem yang bekerja dapat dikelompokkan dalam beberapa bagian yaitu *front-end server*, dan *back-end server*. *front-end* dapat dikatakan sebagai sebuah *gateway* yang menghubungkan akses luar jaringan ke dalam jaringan *back-end*. dalam desain ini *back-end* adalah sistem replikasi *storage* raft. *front-end* menerima *request* dari *client* dan meneruskannya ke *back-end* untuk diproses.

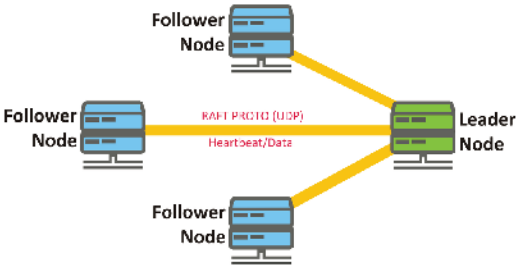
Secara garis besar sistem replikasi pada penelitian ini menggunakan *privat network* dimana setiap *instance* saling mereplikasi data dalam satu alamat jaringan saja. *Front-end server* berlaku sebagai *controller/manager* untuk *key* dan *value* yang akan dilakukan replikasi, *front-end server* juga digunakan sebagai *gateway* antara *client* dengan *raft cloud*.

Seperti Gambar 3.1, garis besar dari sistem yang akan dibangun mempunyai tiga buah komponen dasar dan mempunyai protokol komunikasi yang berbeda. Komunikasi *client host* ke *front-end host* menggunakan *HTTP (hyper text transport protocol)*, karena berupa *storage server* dan data yang akan disimpan adalah *binary* data maka menggunakan protokol *HTTP POST*. Sedangkan komunikasi dari *front-end host* ke raft *cloud* menggunakan protokol sendiri menggunakan *UDP*, penjelasan tentang struktur data protokol tersebut akan dijelaskan di sub-bab 3.1.2.



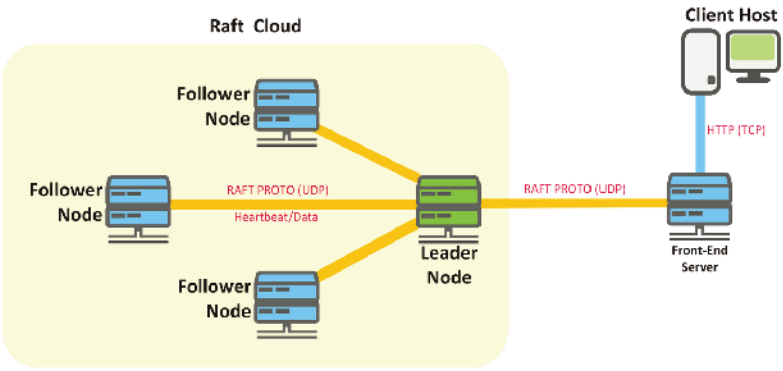
Gambar 3.1: Garis besar desain sistem

Dengan desain seperti Gambar 3.1 maka hanya *host* yang mengetahui protokol komunikasi raft *cloud* saja yang dapat berkomunikasi dengan sistem replikasi *storage server* raft, sistem replikasi raft akan selalu melakukan identifikasi setiap paket data yang masuk melalui *socket UDP*. Oleh karena itu pembuatan aplikasi atau *service* pada *front-end host* harus menggunakan *API (Application Programming Interface)* yang sama digunakan oleh raft cloud.



Gambar 3.2: Komunikasi antar *node* pada raft *cloud*

Setiap data komunikasi pada sistem replikasi baik *heartbeat* maupun pertukaran data seperti Gambar 3.2 menggunakan protokol khusus, sehingga jika ada suatu *host* yang tidak terdaftar namun berada dalam jaringan lokal akan mendapatkan respon *error* saat mengirimkan data.



Gambar 3.3: Desain keseluruhan sistem

Gambar 3.3 merupakan desain keseluruhan sistem yang akan dibangun pada penelitian ini, namun bagian yang paling penting adalah sistem replikasi raft saja, bagian lain seperti *front-end server* hanya tambahan yaitu sebagai manajemen *key* untuk menyimpan sebuah *file*. Jika sistem replikasi raft sudah dapat menyimpan data *binary* (dalam hal ini dapat berupa blok data *bytes*) maka penelitian ini sebenarnya sudah bisa dianggap selesai.

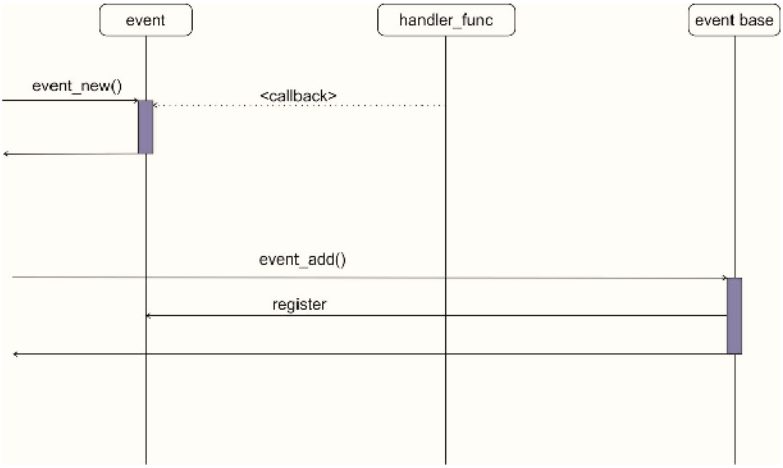
3.1.1 Asynchronous Non-blocking Raft

Dalam penelitian ini bahasa pemrograman yang digunakan adalah *C* dan menggunakan paradigma *event-driven programming*. *event-driven programming* adalah sebuah paradigma pemrograman dimana alur program ditentukan oleh sebuah *event* atau aksi seperti mouse klik, sentuhan tombol, *timeout*, perubahan file, *I/O socket*, dan lain-lain. Dalam sebuah aplikasi *event-driven* terdapat *main-loop* atau biasa disebut dengan *looper* yang melakukan *listen* untuk sebuah *event*, jika ada sebuah *event* maka *looper* akan memicu fungsi *callback*. *Event* yang digunakan pada penelitian ini berasal dari paket data yang masuk ke *socket descriptor* selain itu *timeout* juga digunakan sebagai *event*.

Dalam membuat aplikasi *event-driven* dan menggunakan *asynchronous non-blocking I/O* maka dibutuhkan *library* penunjang salah satunya adalah *libevent*[10]. *Libevent* sendiri mempunyai bebe-

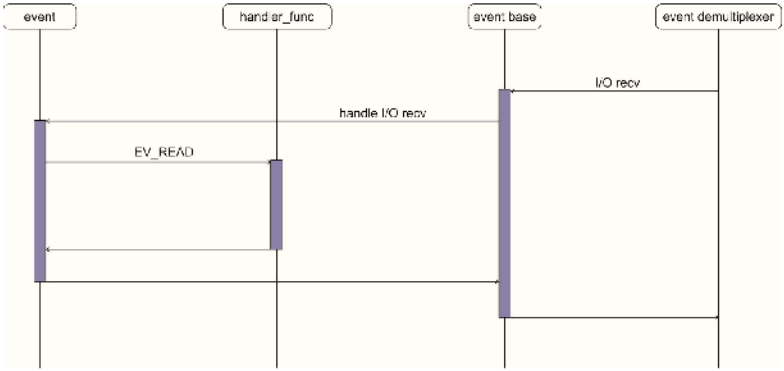
rapa komponen utama yaitu :

1. *Libevent base (looper)*
2. *Libevent event*
3. *Event demultiplexer (pool, epool, select)*



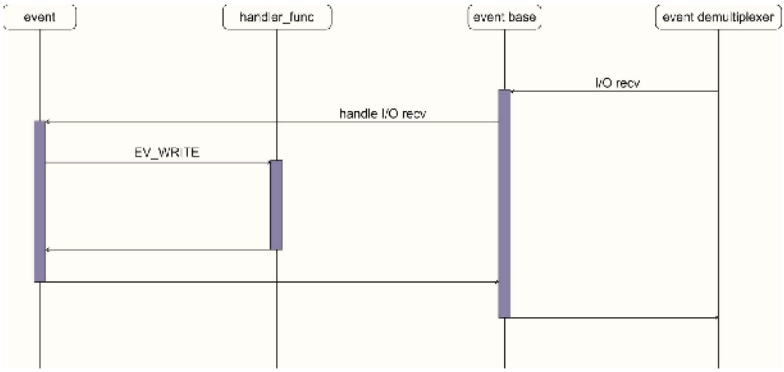
Gambar 3.4: Proses penetapan sebuah *event* ke dalam *looper*

Sequence diagram 3.4 menjelaskan mekanisme penetapan *event* yang akan di respon oleh *looper*. Aplikasi akan menerima *event* dan memicu *callback*, aplikasi dapat mengganti *event* yang akan ditangani dengan cara memanggil fungsi *event_add()* kemudian melakukan pendaftaran.



Gambar 3.5: Proses jika terdapat paket data masuk

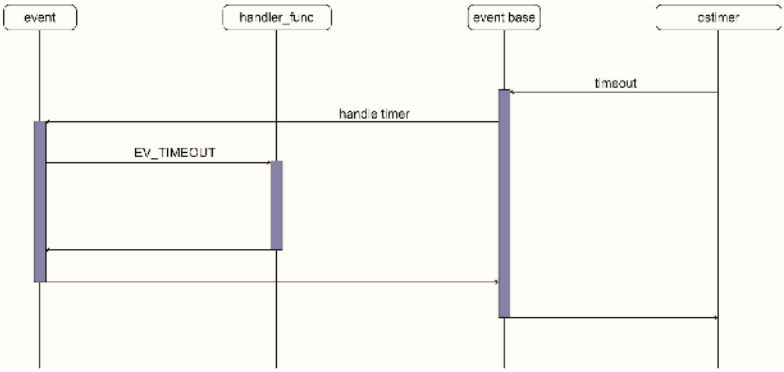
Sequence diagram 3.5 menjelaskan jika terdapat sebuah data masuk dalam *socket descriptor* hal ini dilakukan oleh *pool/epool* di sisi *kernel*, *event looper* akan melakukan penanganan dan memberikan tanda sebagai *EV_READ* yang kemudian akan ditangani oleh fungsi *callback*.



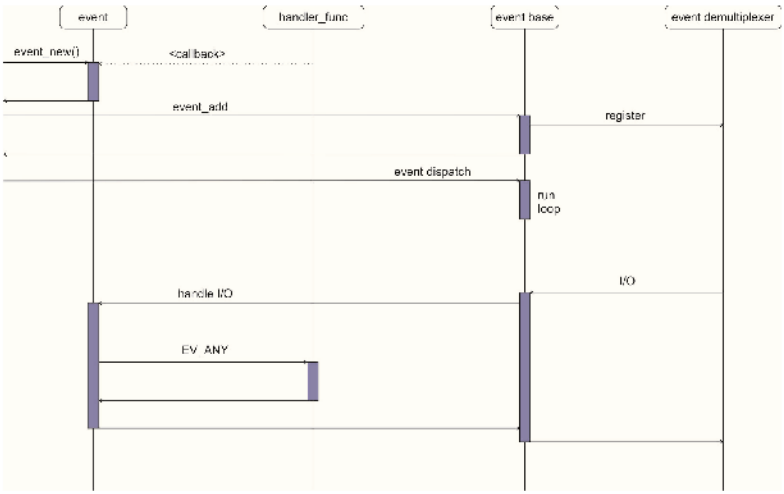
Gambar 3.6: Proses jika socket siap melakukan write data

Proses *write* data, dalam kasus ini mengirim data melalui *UDP socket* dapat dilakukan menggunakan *event* yaitu *EV_WRITE*, Gambar 3.6 menjelaskan bagaimana proses *write* data dimana *writing*

data dilakukan jika *socket* telah siap melakukan *writing*.



Gambar 3.7: Proses jika terjadi *timeout*



Gambar 3.8: Proses keseluruhan implementasi menggunakan *asynchronous non-blocking I/O*

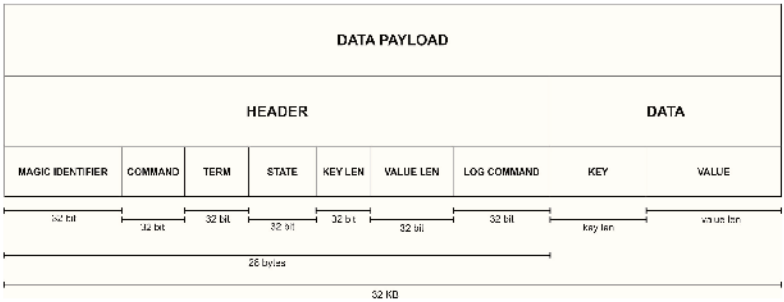
Dalam penelitian ini juga digunakan *event* berupa *timeout*, *timeout* diambil dari *OS timer* dan *looper* akan menangani serta

menandai sebagai *EV_TIMEOUT*, *event* kemudian ditangani oleh fungsi *callback*. proses ini dijelaskan pada Gambar 3.7.

Kesuluruhan proses dalam aplikasi replikasi raft pada penelitian ini dijelaskan pada Gambar 3.8. Pada Gambar 3.8 dijelaskan proses penetapan *event*, kemudian melakukan event *dispatch* yaitu menjalankan *looper* sesuai dengan *event* yang telah ditetapkan, setelah itu melakukan pembacaan *event* data dari *event demultiplexer*.

3.1.2 Desain Struktur Data Komunikasi Replikasi

Pengiriman paket data antar *node* raft baik data *heartbeat* maupun data replikasi menggunakan struktur data sendiri. Alur dari aplikasi akan mengikuti perintah yang ada pada setiap paket data. Semua data yang akan dikirim akan dimasukkan ke dalam struktur data *payload* terlebih dahulu, di sisi penerima akan melakukan *parsing payload*, kemudian membaca *header* sebelum membaca data.



Gambar 3.9: Struktur data *payload*

Gambar 3.9 adalah struktur data *payload* sebagai komunikasi antar *node* dalam sistem replikasi raft. Setiap paket *payload* harus mempunyai *magic identifier* yang telah disepakati sepanjang 32 *bit*, jika terdapat paket data dimana 32 *bit* pertama tidak sama dengan *magic identifier* yang telah disepakati maka akan mendapatkan respon *error* atau tidak mendapatkan respon sama sekali.

```
typedef enum {
  CMD_NONE = 0,
  CMD_HEARTBEAT_SEND,
  CMD_HEARTBEAT_OK,
  CMD_VOTE_REQ,
  CMD_VOTE_GRANT,
  CMD_VOTE_REJECT,
  CMD_SUPER_COMMAND,
  CMD_DATA_IN,
  CMD_LOG_REPLICATE,
  CMD_LOG_REPLICATE_OK,
  CMD_LOG_COMMIT,
  CMD_GET_LOG_SYNCHRONIZE,
  CMD_SEND_LOG_SYNCHRONIZE,
  CMD_READ_DATA,
  CMD_READ_DATA_OK,
  CMD_READ_DATA_ERR,
  CMD_WHOIS_LEADER
} raft_command;
```

Kode 3.1: Enumerasi raft *command*

32 *bit* setelah *magic identifier* adalah raft *command* dimana menentukan perintah dari paket tersebut, pada implementasinya sistem pada penelitian ini menggunakan enumerasi tipe data *integer*, *listing* kode 3.1 adalah *listing* kode untuk raft *command*. Penjelasan dari perintah-perintah tersebut ada pada tabel 3.1.

Tabel 3.1: Keterangan raft *command*

Perintah/Command	Keterangan
CMD NONE	Tidak ada perintah apapun, digunakan pada saat inisialisasi raft saja
CMD HEARTBEAT SEND	digunakan oleh *node leader* untuk mengirimkan paket data *heartbeat*
CMD HEARTBEAT OK	digunakan oleh *follower* untuk membalas paket *heartbeat*

CMD VOTE REQ	digunakan oleh *node* dengan status kandidat, perintah untuk *request voting* pada saat pemilihan *leader*
CMD VOTE GRANT	perintah untuk menyetujui *voting*
CMD VOTE REJECT	perintah untuk tidak menyetujui *voting*
CMD SUPER COMMAND	digunakan oleh *leader*, setiap *node* yang mendapat perintah ini akan merubah statusnya menjadi *follower*
CMD DATA IN	perintah yang diterima oleh *leader* setiap ada data dari *front-end server*
CMD LOG REPLICATE	digunakan oleh *leader* untuk melakukan replikasi data serta *log*
CMD LOG REPLICATE OK	jika replikasi sukses diterima dan disimpan oleh *follower*, maka perintah ini digunakan untuk membalas kepada *leader*
CMD LOG COMMIT	digunakan sebagai perintah untuk *commit log* dan *increment state*
CMD GET LOG SYNCHRONIZE	jika state *follower* tertinggal maka perintah ini digunakan untuk meminta data serta log kepada *leader*
CMD SEND LOG SYNCHRONIZE	digunakan untuk mengirim *log* yang tertinggal
CMD READ DATA	untuk mengambil data
CMD READ DATA OK	pesan kembalian jika data dapat dibaca dengan baik

CMD READ DATA ERR	kembalian jika proses pembacaan data error
CMD WHOIS LEADER	hanya *leader* yang akan merespon command ini

Blok data *term* serta *state* pada *header payload* berisi *term* serta *state* dari pengirim paket yang selanjutnya digunakan seperti bagaimana algoritma raft bekerja. jika raft *command* adalah perintah replikasi maka 96 *bit* terakhir dari *header* akan digunakan, *key len* mempunyai nilai *integer* untuk membaca data berupa *key* sepanjang nilai dari *key len*, sedangkan *value len* digunakan untuk membaca data *value*. *Log command* adalah untuk melihat perintah replikasi seperti *write*, *update* atau *delete*.

Panjang maksimal data *payload* sebenarnya dapat sepanjang 65507 *bytes* yaitu didapat dari :

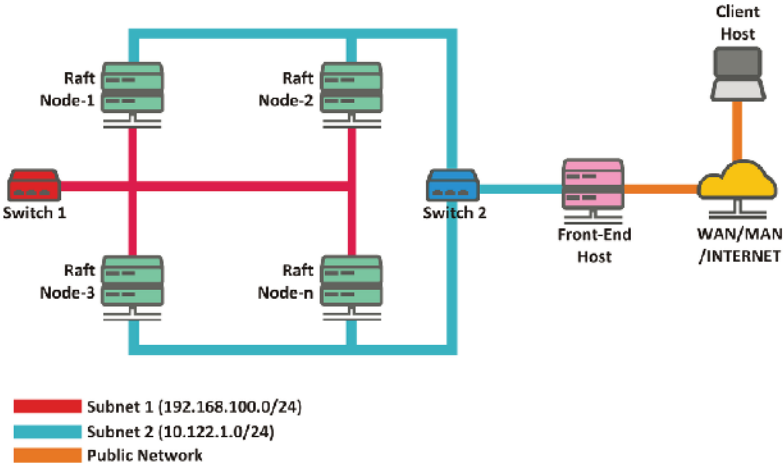
$$\begin{aligned} MAX &= MAXIPpacket - IPheader - UDPheader \\ MAX &= 65535 - 20 - 8 \\ MAX &= 65507 \end{aligned} \tag{3.1}$$

Namun pada penelitian ini digunakan sepanjang 32768 *bytes* atau 32KB saja karena pada beberapa sistem operasi melakukan limitasi pada toleransi maksimal transmisi paket IP yaitu 32KB. *Payload* sepanjang 32KB ini sebenarnya masih dilakukan fragmentasi sesuai dengan *MTU Ethernet* yaitu sebesar 1500 *bytes*, namun pada penelitian ini fragmentasi tersebut diabaikan karena proses replikasi menggunakan jaringan lokal sehingga dapat ditoleransi.

3.2 Desain Jaringan

Setiap komputer pada sistem yang dibuat di penelitian ini mempunyai dua buah *ethernet card* karena setiap komputer akan mempunyai dua alamat jaringan yang berbeda.

Seperti pada Gambar 3.10, sebagai contoh pada alamat jaringan lokal untuk replikasi mempunyai alamat *IP* 192.168.100.0/24, karena menggunakan protokol *UDP* maka replikasi menggunakan



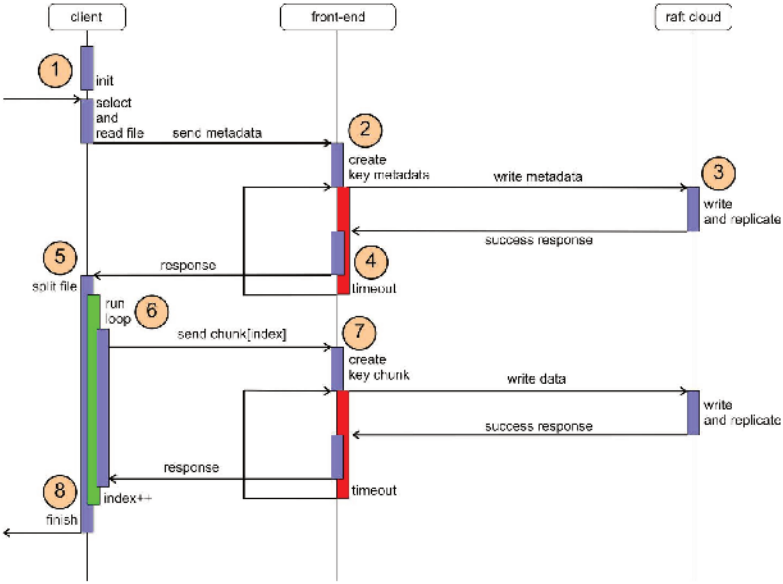
Gambar 3.10: Topologi jaringan replikasi

alamat jaringan lokal yang tidak terganggu dengan aktivitas jaringan lain. Sedangkan di sisi lain *node* raft yang terhubung dengan *front-end* mempunyai alamat jaringan 10.122.1.0/24 dimana replikasi tidak terjadi di alamat jaringan ini.

3.3 Alur Implementasi Sistem

Implementasi sistem dibagi menjadi tiga bagian yaitu alur penyimpanan data, pengambilan data, serta penghapusan data. Semua langkah baik menyimpan, mengambil, maupun menghapus data dilakukan oleh *webclient* menggunakan protokol *HTTP*. *Aplikasi* web yang digunakan oleh *client* dibangun menggunakan *HTML, CSS dan Javascript*. Namun yang mempunyai peran paling penting disini adalah *javascript* karena proses pemecahan file menjadi *chunk-chunk* yang kecil hanya bisa dilakukan oleh *javascript* dalam implementasi aplikasi web. Sementara itu karena menggunakan *key-value model* maka proses manajemen *key-value* terjadi di sisi *front-end host*.

3.3.1 Alur Penyimpanan Data



Gambar 3.11: Alur proses penyimpanan file ke *storage server*

Gambar 3.11 merupakan *sequence diagram* yang menjelaskan alur penyimpanan sebuah file ke *storage server*. Karena *storage server* menggunakan *key-value model* maka metode penyimpanan adalah berupa penyimpanan *chunk* file yaitu file yang semula berukuran besar akan dipotong menjadi beberapa bagian kecil dan dilakukan penyimpanan tiap *chunk*. Setiap *chunk* mempunyai *key* yang berbeda sehingga proses replikasi yang dilakukan pada penelitian ini sebenarnya adalah replikasi dari *chunk* file.

Penjelasan dari alur implementasi *Sequence diagram* 3.11 adalah sebagai berikut :

1. Pada implementesinya *client* adalah aplikasi web dimana *user* mempunyai identitas yang unik. Identitas yang unik ini dapat berupa *username, user id, access token* dan lain sebagainya

2. Setelah identitas user dipastikan, user dapat memilih file yang akan disimpan di *server*. Langkah pertama adalah mengirimkan *metadata* file tersebut seperti nama file, ukuran file, serta *owner* file. Pembentukan *key* dari *metadata* ini adalah

$$key = userid.timemillis.filename.meta$$

karena identitas *user* adalah unik maka *key* untuk *metadata* file juga bersifat unik. Setiap user mempunyai kemungkinan untuk menyimpan data yang sama, maka digunakan *time milliseconds* pada saat menentukan *key file* tersebut untuk mencegah konflik pada *key*

3. Penyimpanan data serta replikasi, data dikirimkan kepada *leader raft cloud* kemudian dilakukan replikasi sesuai dengan algoritma raft. Jika replikasi sukses maka akan mengirimkan respon sukses

4. *Timeout* terjadi jika sistem raft tidak mengirimkan respon selama jangka waktu tertentu, jika terjadi *timeout* maka akan mengirimkan data tersebut satu kali lagi

5. *Client* menerima respon dari *server* bahwa *metadata* sukses disimpan, hal yang dilakukan *client* selanjutnya adalah membagi file menjadi beberapa bagian yang disebut dengan *chunk*

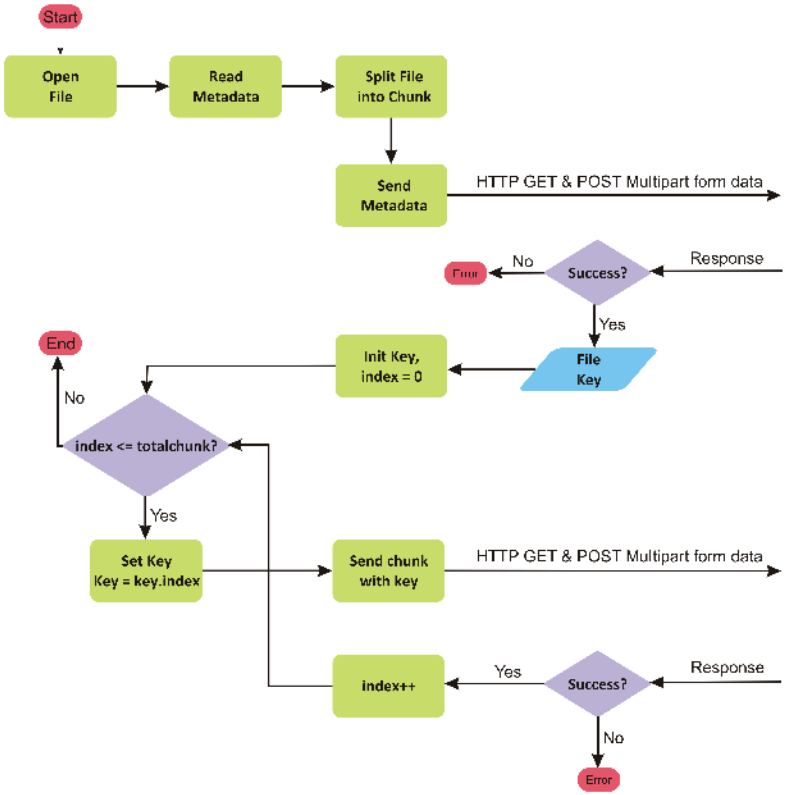
6. *Client* melakukan perulangan sebanyak jumlah *chunk* untuk mengirimkan *chunk* satu per satu

7. Membentuk *key* untuk *chunk* file, *key* dari *chunk* tersebut adalah

$$key = userid.timemillis.filename.index$$

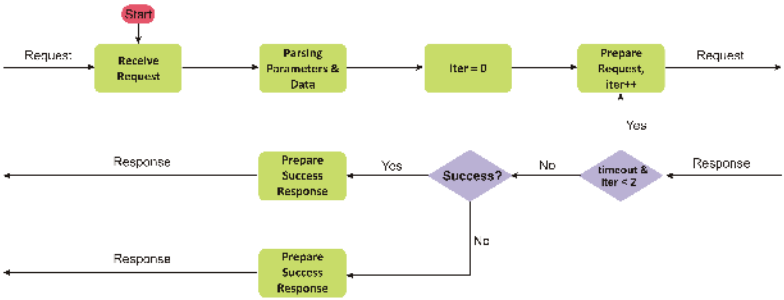
data *key* serta *value* berupa *chunk* file kemudian akan dikirim ke sistem raft untuk dilakukan penyimpanan dan replikasi

8. Pengiriman selesai, *user* akan menerima data berupa informasi untuk mengambil file tersebut



Gambar 3.12: Alur proses penyimpanan file sisi *client*

Untuk memperjelas alur penyimpanan data pada sisi *client* dapat dilihat pada Gambar 3.12. Pada intinya client akan melakukan *request* untuk menyimpan metadata terlebih dahulu. Setelah *request* berhasil dilakukan dan mendapat respon sukses maka langkah selanjutnya melakukan pengiriman data *chunk* secara berurutan menggunakan *key* yang didapatkan pada saat menyimpan metadata. Pengiriman data menggunakan *HTTP POST* serta *GET* sekaligus. *HTTP GET* digunakan untuk mengirim *query* data *key*, sedangkan *HTTP POST* untuk mengirim *chunk data*.



Gambar 3.13: Alur proses penyimpanan file sisi *front-end host*

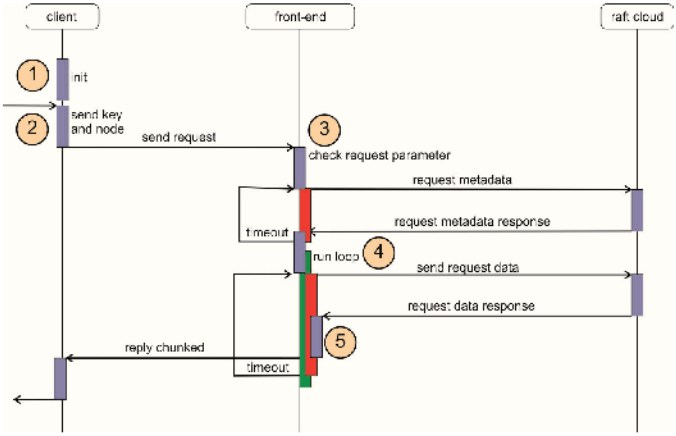
Pada sisi *front-end host* dijelaskan lebih detail pada Gambar 3.13. Pada proses penyimpanan data ini *front-end host* hanya menerima data serta parameter yang kemudian dikirimkan ke *raft cloud* sesuai struktur data yang yang disepakati.

3.3.2 Alur Pengambilan Data

Pengambilan data dilakukan dengan hal yang hampir sama dengan penyimpanan data, namun untuk pengambilan data proses *loop* terjadi di sisi *front-end server* karena melakukan pengambilan data tiap *chunk* dari raft cloud.

Jika proses penyimpanan data mengirimkan data dan menghasilkan sebuah *key*, maka proses pengambilan data adalah sebaliknya. Pada proses pengambilan data hal yang paling penting adalah *key* yang kemudian digunakan untuk mengambil data pada *raft cloud*. *Key* harus benar - benar valid karena jika tidak maka data tidak akan ditemukan.

Karena menggunakan protokol *HTTP* maka informasi seperti nama file, *mime-type* serta ukuran file diperlukan pada saat mengambil data (*downloading*). Informasi tersebut dicantumkan pada *HTTP header* sebagai informasi penanda untuk *client*. Oleh karena itu proses yang pertama kali dilakukan adalah mengambil metadata yang kemudian akan dicantumkan pada *HTTP header* untuk segera dikirimkan kepada *client*. Transfer file pada proses ini menggunakan *Transfer-encoding : chunked*.



Gambar 3.14: Alur proses pengambilan file dari *storage server*

Seperti digambarkan pada *Sequence diagram* 3.14 *client* hanya melakukan satu kali *http request* saja, seluruh chunk data akan diambil oleh front-end server. Detil penjelasannya adalah :

1. Inisialisasi *client* dengan membuka file html sebagai *interface* untuk melakukan pengambilan data.

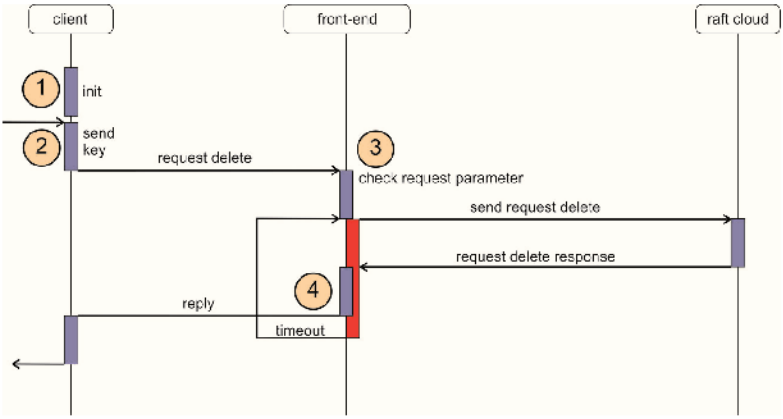
2. Mengirimkan key serta *IP Address* dari node yang diinginkan untuk melakukan pengambilan data.

3. Melakukan *parameter checking*, jika parameter tidak lengkap atau parameter tidak bisa dilakukan *parsing*, maka *request* tidak akan diproses

4. Mendapatkan *metadata* dan membacanya untuk melakukan request chunk ke *raft cloud* serta memberikan *http response header* kepada *client*

5. Mengirimkan *header* yang berisi info koneksi serta beberapa info dari data seperti nama file, ukuran file, dan tipe file. setelah mengirimkan *header* maka yang dilakukan adalah mengirimkan *chunk* data satu per satu.

3.3.3 Alur Penghapusan Data



Gambar 3.15: Alur proses hapus file *storage server*

Pada dasarnya menghapus data hanya menghapus metadata saja, dengan cara menghapus metadata maka data tidak dapat ditemukan melalui perlakuan yang normal. Karena metadata tidak ditemukan maka sistem akan menganggap data tidak ditemukan. Beberapa proses pada *Sequence diagram* 3.15 adalah :

1. Inisislisasi *client* dengan cara membuka aplikasi web untuk *delete*
2. Mengirimkan key dari data yang akan dihapus
3. Melakukan cek parameter
4. Hapus data selesai, dikembalikan ke *client*

Halaman ini sengaja dikosongkan

BAB 4
PENGUJIAN DAN ANALISIS

Pada bab ini pengujian dibagi menjadi beberapa metode yaitu pengujian *leader election* serta validasi data (*read and write*). Pengujian dilakukan dengan cara mengidentifikasi hasil *md5sum* dari file sebelum dilakukan penyimpanan ke *storage server* dan setelah diambil dari *storage server*.

4.1 Perangkat Keras dan Sistem Operasi

Komputer yang dipakai dalam pengujian penelitian ini sejumlah empat unit, tiga unit sebagai *raft cloud* dan satu unit sebagai *front-end server* seluruhnya mempunyai spesifikasi yang sama.

Tabel 4.1: Spesifikasi dari *raft node*

Spesifikasi	Keterangan
Jumlah Prosesor	1
Jumlah Core	Quad-core
Kecepatan Prosesor	3.46 GHz
Jenis Prosesor	Intel® Xeon® 5600 series
Cache	8 MB L3
Arsitektur	x86_64
Memori	6 GB
Disk Storage	300 GB
Power Supply	460 Watt
Sistem Operasi	Ubuntu 16.04.2 LTS GNU-/Linux 4.4.0-66-generic

Selain menggunakan empat buah komputer seperti pada Tabel 4.1, digunakan juga sebuah *Single Board Computer (SBC)* yaitu *Raspberry-Pi* sebagai *webserver* untuk aplikasi web yang akan digunakan menjadi *client*. Selain itu juga digunakan dua buah *switch* karena sistem ini membutuhkan dua alamat jaringan.

Tabel 4.2: Informasi *switch* yang digunakan dalam pengujian

No	Vendor	Tipe	Alamat Jaringan
1	Mikrotik	10/100 Mbps Managed	192.168.100.0/24
2	TP-Link	10/100 Mbps Unmanaged	10.122.1.0/24

Dapat dilihat pada Tabel 4.2 bahwa sistem ini menggunakan dua buah *switch*. *Switch* yang pertama mempunyai kecepatan maksimal 1000 *Mbps* atau biasa disebut *gigabit switch*. *Switch* ini menangani alamat jaringan 192.168.100.0/24 yang digunakan sebagai replikasi data. *Switch* yang kedua mempunyai spesifikasi lebih rendah dan menangani alamat jaringan 10.122.1.0/24 dimana digunakan oleh *client* untuk berhubungan dengan *front-end host*.

Tabel 4.3: Informasi setiap *instance* sistem

No	Peran	Tipe	Alamat IP 1	Alamat IP 2
1	*Raft NODE-1*	Server 3U	192.168.100.12	10.122.1.212
2	*Raft NODE-2*	Server 3U	192.168.100.14	10.122.1.214
3	*Raft NODE-3*	Server 3U	192.168.100.15	10.122.1.215
4	*Front-end Host*	Server 3U	192.168.100.10	10.122.1.210
5	*Webserver*	SBC	-	10.122.1.39

Tabel 4.3 adalah informasi peran tiap *instance* serta alamat yang digunakan. Semua menggunakan komputer yang sama kecuali *webserver* yang menggunakan *SBC* karena hanya melayani proses yang cukup ringan. *Raft node* serta *front-end host* adalah kesatuan sistem yang tidak dapat dipisahkan sehingga harus mempunyai alamat jaringan atau *subnet* yang sama.

4.2 Pengujian Leader Election

Pengujian *leader election* dilakukan dengan cara mematikan *node leader* dan melihat apakah *leader* baru akan terpilih. Pengujian ini dilakukan untuk mengetahui apakah sistem dapat melakukan pemilihan *leader* dengan baik dan tidak ada konflik. Konflik akan terjadi jika terdapat lebih dari satu leader pada satu sistem.

Untuk mengetahui status dari tiap *raft node* maka pengguna dapat melihat melalui halaman web untuk *monitoring*. Halaman web untuk *monitoring node* dapat diakses melalui *web browser* pada alamat *http://10.122.1.39/raft/monitor/*.

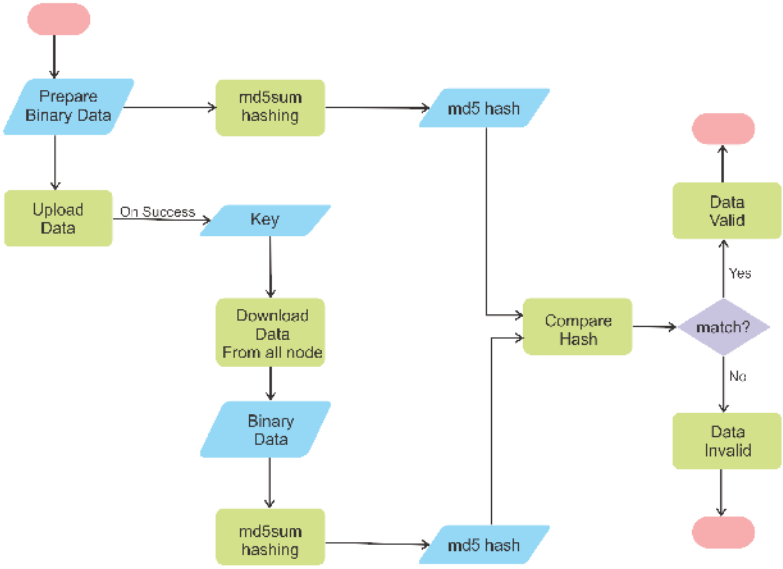
Tabel 4.4: Hasil uji *leader election*

No	Aksi	Status		
		node-1	node-2	node-3
1	*start up*	*leader*	*follower*	*follower*
2	*node-1 down*	*down*	*leader*	*follower*
3	*node-1 up*	*follower*	*leader*	*follower*
4	*node-2 down*	*leader*	*down*	*follower*
5	*node-2 up*	*leader*	*follower*	*follower*
6	*node-1 down*	*down*	*follower*	*leader*
7	*node-3 down*	*down*	*candidate*	*down*
8	*node-3 up*	*down*	*follower*	*leader*
9	*node-3 down*	*down*	*down*	*candidate*
10	*node-1 up, node-2 up*	*follower*	*follower*	*leader*

Hasil pengujian *leader election* dapat dilihat pada Tabel 4.4, Proses pengujian dilakukan secara berurutan dengan melakukan aksi secara acak, berawal dari sistem *start up* dimana *node-1* berlaku sebagai *leader* hingga aksi terakhir dimana posisi *leader* berpindah pada *node-3*. Dapat terlihat dari hasil pengujian bahwa setiap aksi yang dilakukan tidak terdapat lebih dari satu *leader*, hal ini telah sesuai dengan algoritma konsensus raft dimana dalam satu sistem tidak bisa menjalankan dua *leader*.

4.3 Uji Validasi Data

Uji validasi data dilakukan dengan cara melakukan penyimpanan file ke *storage server* dengan berbagai ukuran. Namun sebelum melakukan penyimpanan dilakukan *md5sum hashing* pada file tersebut. seteleh penyimpanan berhasil, maka dilakukan pengambilan file melalui semua *node* raft satu per satu. Data yang telah diambil akan dilakukan *md5 hash* satu per satu apakah sama dengan file aslinya. Skenario pengujian ini dapat dilihat pada Gambar 4.1.



Gambar 4.1: Skenario pengujian validasi data

Tabel 4.5: Hasil uji *md5sum* dengan menampilkan 5 karakter pertama dan terakhir *md5 hash*

No	Ukuran File	md5 hash	Download dari			Keterangan
			node 1	node 2	node 3	
1	10 MB	f3460...6e905	f3460...6e905	f3460...6e905	f3460...6e905	*md5sum valid*
2	50 MB	8c15b...c8d49	8c15b...c8d49	8c15b...c8d49	8c15b...c8d49	*md5sum valid*
3	100 MB	3a1d3...76bb6	3a1d3...76bb6	3a1d3...76bb6	3a1d3...76bb6	*md5sum valid*
4	500 MB	31db1...73051	31db1...73051	31db1...73051	31db1...73051	*md5sum valid*
5	1 GB	02ee8...82179	02ee8...82179	02ee8...82179	02ee8...82179	*md5sum valid*

Pengujian dilakukan beberapa kali dengan ukuran file yang berbeda - beda. Mulai dari file berukuran kecil dibawah 100 MB, file berukuran sedang, hingga file berukuran besar yaitu berukuran 5 GB. file yang digunakan adalah file dengan data *random* yang dibuat dengan *bash scripting* seperti pada Kode 4.1.

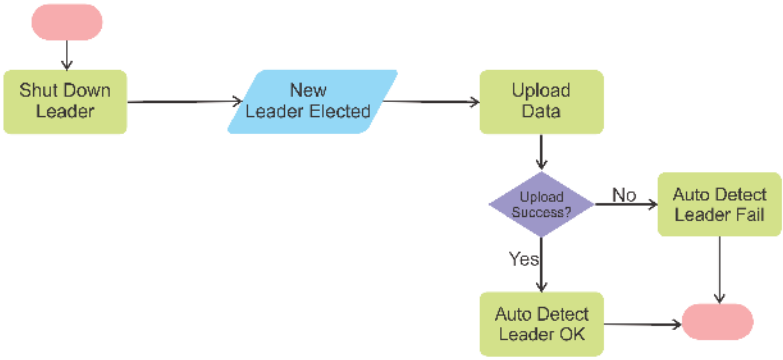
```
#/bin/bash
SIZES=('10M' '50M' '100M' '500M' '1000M' '5000M');
for i in `seq 0 $((${#SIZES[@]}-1))`; do
  head -c ${SIZES[$i]} < /dev/urandom > ${SIZES[$i]}.data
done
```

Kode 4.1: Script untuk membuat file *random*

Dari hasil uji pada tabel 4.5, semua data hasil *download* dari tiap *raft node* mempunyai hasil *md5sum* yang sama. Dengan beberapa kali percobaan dengan ukuran file berbeda - beda tersebut maka dapat disimpulkan bahwa tidak ada blok data yang hilang karena uji *md5sum* mempunyai nilai yang sama dengan kata lain tidak ada penambahan atau pengurangan data sama sekali.

4.4 Uji Deteksi Leader Secara Otomatis

Proses penyimpanan data pertama kali dilakukan oleh *leader node* sehingga seluruh data dari *client* akan dikirimkan ke *raft leader* oleh *front-end host*. Permasalahan yang terjadi adalah jika status *leader* berubah karena terjadi suatu kesalahan pada *leader node* yang sebelumnya. Untuk menanggulangi permasalah tersebut maka *front-end host* harus mengetahui dengan pasti dimana *leader host* berada sebelum melakukan pengiriman data. Pengujian ini sesuai dengan skenario yang digambarkan pada Gambar 4.2.

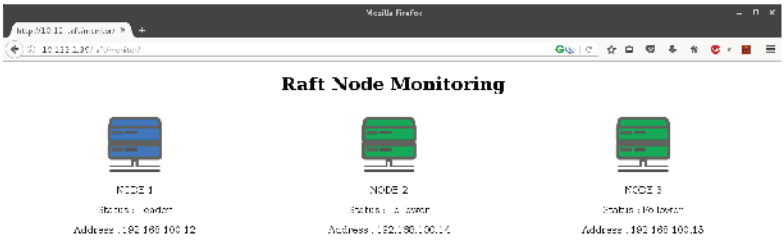


Gambar 4.2: Skenario uji deteksi leader

Seperti pada Gambar 4.2 proses pengujian bermula dengan melihat dimana leader berada, setelah mengetahuinya maka langkah yang dilakukan adalah mematikan node leader sehingga leader lain akan terpilih. Tanpa melakukan konfigurasi ulang pada *front-end host*, client langsung melakukan *upload* data setelah node semua node diaktifkan kembali. Jika *client* berhasil melakukan *upload* data, maka sistem dapat melakukan deteksi *leader* secara otomatis.

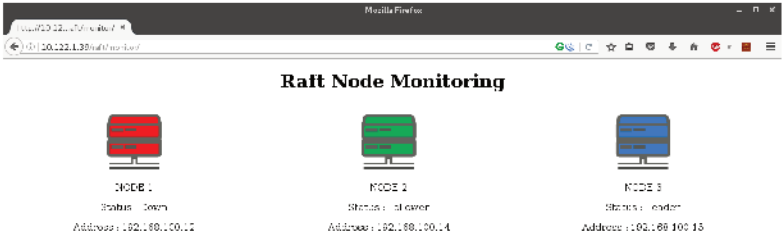
4.4.1 Proses Pengujian

Proses pengujian akan dilakukan sesuai dengan alur pada Gambar 4.2. Langkah-langkah pengujian akan dijelaskan dengan gambar untuk memperlihatkan prosesnya. Pengujian ini dilakukan dengan cara mengetahui posisi *leader* terlebih dahulu dengan cara mengakses halaman web *monitoring*. Jika suatu *node* berlaku sebagai *leader* maka akan menampilkan warna biru pada ikon *node*, jika *node* adalah *follower* atau *candidate* akan menampilkan warna hijau, sedangkan warna merah menandakan *node* tersebut tidak aktif atau *down*.



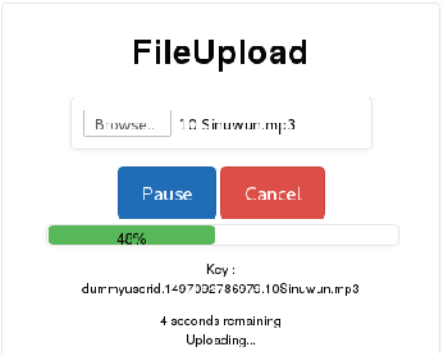
Gambar 4.3: Awal mula status *node*

Pada awal mula leader berada pada *node-1*, *node* lain berlaku sebagai *follower*. Informasi tersebut dapat dilihat pada web monitor seperti pada Gambar 4.3.



Gambar 4.4: Status *node* setelah *node-1* dimatikan

Setelah status tiap *node* diketahui, maka langkah selanjutnya adalah melakukan *shutdown leader* dalam kasus ini *leader node* adalah *node-1*. Langkah tersebut dapat dipastikan dengan melihat web monitor seperti yang terlihat pada Gambar 4.4 dimana *node-1* yang semula berlaku sebagai *leader* berwarna merah, *node-2* sebagai follower dan posisi *leader* berpindah pada *node-3* dangan tampilan ikon berwarna biru.



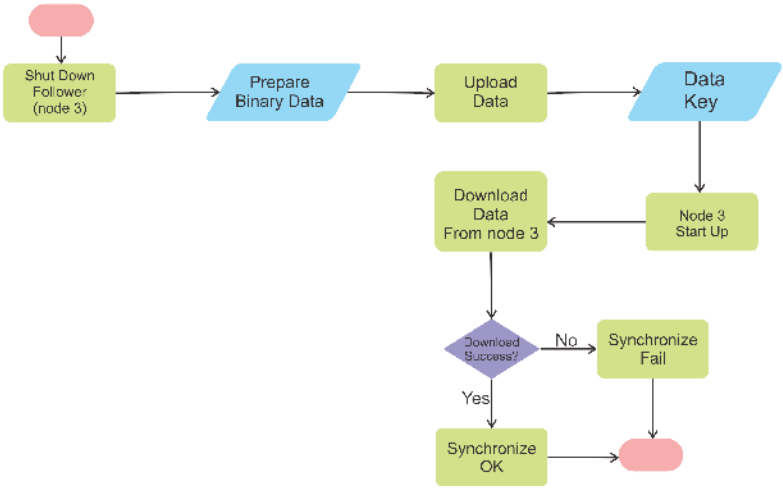
Gambar 4.5: Client berhasil melakukan *upload* data

Pada posisi tersebut client melakukan *upload* data, jika berhasil melakukan *upload* data maka sistem dapat menentukan secara otomatis dimanakah *leader node* berada. Pengujian dapat dilihat pada Gambar 4.5 dimana *client* sedang dalam proses melakukan *upload* file dan telah mendapatkan *key* dari file tersebut. Informasi tersebut menunjukkan bahwa *client* berhasil melakukan proses *upload* dengan baik.

Setelah melakukan proses ini maka dapat disimpulkan bahwa sistem dapat menentukan *leader* secara otomatis. Hal itu ditunjukkan oleh keseluruhan proses percobaan dimana *client* berhasil melakukan *upload* data setelah *leader* berpindah. Proses *upload* data hanya dapat diterima oleh *leader node* saja yang kemudian dilakukan replikasi ke seluruh *follower*.

4.5 Uji Sinkronisasi Data

Sinkronisasi terjadi pada saat *follower node* mempunyai *state* yang tertinggal dari *node* lain. Hal ini dapat terjadi jika *follower node* tersebut mengalami *system down* pada saat replikasi data terjadi. Oleh karena itu pengujian ini menjadi hal yang sangat penting, proses pengujian sinkronisasi ini dapat dilihat pada Gambar 4.6.



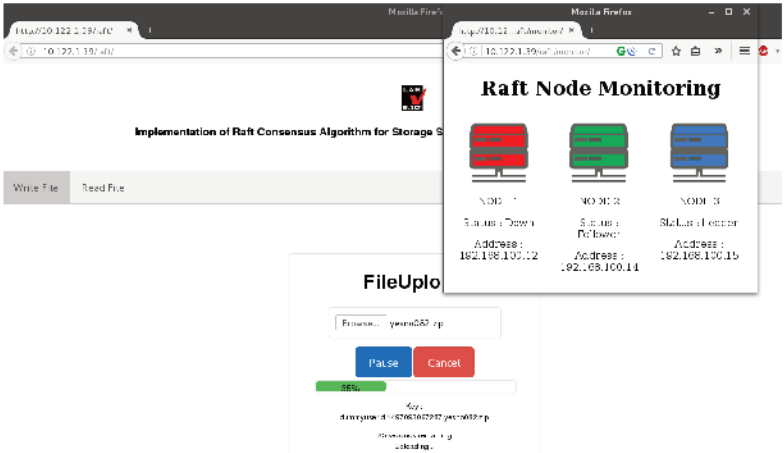
Gambar 4.6: Skenario pengujian sinkroniasasi data

Gambar 4.6 menjelaskan proses pengujian ini. Langkah pertama adalah dengan cara melakukan *shutdown follower* dalam pengujian kali ini adalah *node-1*. Setelah *node-1 down* client akan melakukan *upload* data seperti proses yang seharusnya. Jika proses *upload* telah selesai maka *node 1 up* kembali, client akan *download* dari *node-1* untuk memastikan apakah data dapat diambil. Jika data dapat diambil dengan baik maka dapat disimpulkan bahwa data telah tersinkronisasi dengan baik.

Jika proses pengujian ini gagal dimana data tidak tersinkronisasi dengan baik maka dapat dilakukan penelitian lebih lanjut untuk mengetahui titik kesalahan yang terjadi. Tetapi jika pengujian ini berjalan dengan baik maka menunjukkan bahwa sistem berjalan dengan lancar sesuai dengan konsep awal.

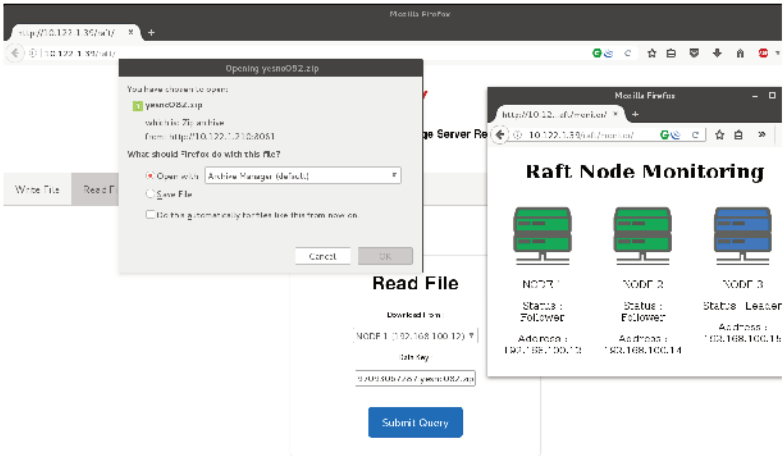
4.5.1 Proses Pengujian

Sesuai dengan algoritma konsensus raft, suatu *node* yang mempunyai *log index* yang tertingal akan secara otomatis melakukan sinkronisasi data terhadap *leader* atau *node* lain yang mempunyai *log index* lebih besar. Pengujian ini dilakukan dengan cara melakukan *shutdown* salah satu *node* untuk membuatnya tertinggal.



Gambar 4.7: Upload data dengan *node 1* down

Gambar 4.7 menunjukkan proses pengujian dimana *client* melakukan *upload* data dengan salah satu *node* yakni *node-1* berada dalam posisi *down*.

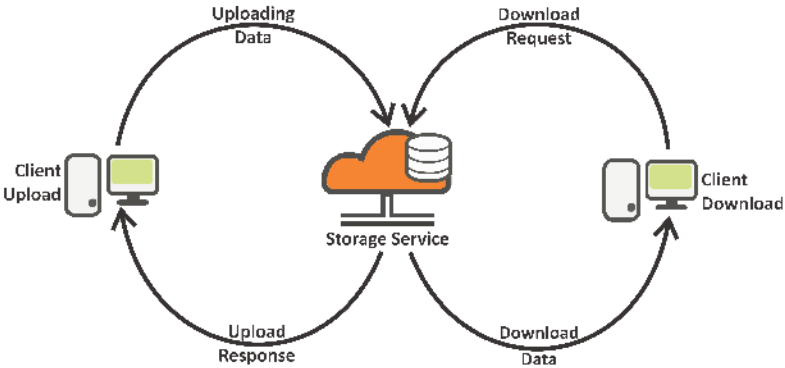


Gambar 4.8: Download data dari *node-1* setelah *node-1* kembali menyala

Setelah *upload* data selesai, maka *node-1* dihidupkan kembali. Ketika *node-1* mulai mendapatkan paket *heartbeat* dari *leader* maka seharusnya *node-1* mengetahui seberapa besar jumlah *log index* yang tertinggal dan mengirim *request* untuk tiap data yang tertinggal.

Proses pengujian dapat dilanjutkan dengan cara *download* data dari *node-1*. Dari pengujian pada Gambar 4.7, client berhasil mendapatkan data dari *node-1*. Dengan informasi ini maka proses pengujian sinkronisasi data dapat berjalan dengan baik sesuai dengan algoritma konsensus raft.

4.6 Uji Download/Stream Pada Saat Upload Berlangsung

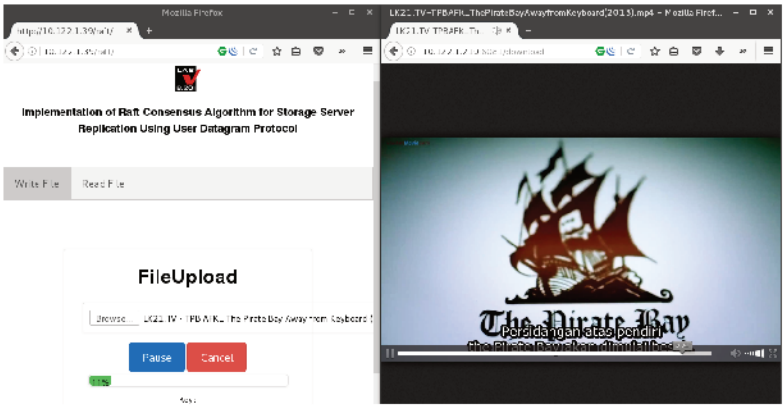


Gambar 4.9: Skenario pengujian upload dan download pada waktu yang bersamaan

Gambar 4.9 menjelaskan proses download serta upload dalam waktu yang bersamaan. Karena penyimpanan data berbentuk *chunked* maka hal ini sangat dimungkinkan. Pada saat *client* melakukan penyimpanan data dan masih dalam proses upload, *client* lain dapat melakukan *download* data sekaligus. Pengujian akan dilakukan dengan file video dengan ukuran besar dimana pada saat salah satu *client* melakukan *upload* video, *client* lain akan melakukan *video streaming*

4.6.1 Proses Pengujian

Proses pengujian dilakukan dengan dua buah *client* dalam implementasi pengujian ini adalah dua buah window *web browser*. *Client* pertama melakukan upload data berupa file video, setelah mulai menjalankan *upload* dan telah mendapatkan *key*, selanjutnya *key* tersebut digunakan untuk *download* data oleh *client* kedua.



Gambar 4.10: Sisi kiri : client melakukan upload data video, kanan : client lain melakukan *streaming* video dengan key yang sama

Pengujian dilakukan seperti *screenshot* pada Gambar 4.10 dimana pada bagian kiri adalah *client* pertama yang sedang melakukan proses upload dan pada bagian kanan adalah *client* kedua yang melakukan proses *download/stream* data video dengan *key* yang sama. Dengan melihat Gambar 4.10 maka dapat disimpulkan bahwa proses *upload* dan *download* dalam waktu yang hampir bersamaan ini dapat dilakukan dengan baik.

4.7 Uji Mengganti Ukuran *Chunk*

Pada awal mula *data chunk* berukuran kecil yaitu sebesar *32KB*. Pada pengujian kali ini akan melakukan penggantian besar *chunk* dan melihat perbedaan waktu *upload*. Pengujian pertama akan mencatat waktu *upload* data dengan ukuran *chunk 32KB*, sedangkan pengujian kedua akan mencatat waktu upload dengan besar *chunk 64KB*.

Hasil dari pengujian ini akan disajikan menggunakan tabel dimana akan mencatat waktu *upload* serta hasil uji validasi data. Data yang digunakan untuk pengujian adalah lima buah file binary dengan ukuran yang berbeda - beda yaitu *10 MB, 50 MB, 100 MB, 500 MB* serta *1GB*.

Tabel 4.6: Hasil uji pencatatan waktu *upload* dengan *chunk* sebesar 32KB

No	Ukuran File	Waktu upload (*ms*)	Download dari		
			node 1	node 2	node 3
1	10 MB	2759	*md5sum valid*	*md5sum valid*	*md5sum valid*
2	50 MB	16819	*md5sum valid*	*md5sum valid*	*md5sum valid*
3	100 MB	28603	*md5sum valid*	*md5sum valid*	*md5sum valid*
4	500 MB	147607	*md5sum valid*	*md5sum valid*	*md5sum valid*
5	1 GB	290111	*md5sum valid*	*md5sum valid*	*md5sum valid*

Tabel 4.7: Hasil uji pencatatan waktu *upload* dengan *chunk* sebesar 64KB

No	Ukuran File	Waktu upload (*ms*)	Download dari		
			node 1	node 2	node 3
1	10 MB	3891	*md5sum invalid*	*md5sum invalid*	*md5sum invalid*
2	50 MB	18559	*md5sum invalid*	*md5sum invalid*	*md5sum invalid*
3	100 MB	42521	*md5sum invalid*	*md5sum invalid*	*md5sum invalid*
4	500 MB	218787	*md5sum invalid*	*md5sum invalid*	*md5sum invalid*
5	1 GB	450739	*md5sum invalid*	*md5sum invalid*	*md5sum invalid*

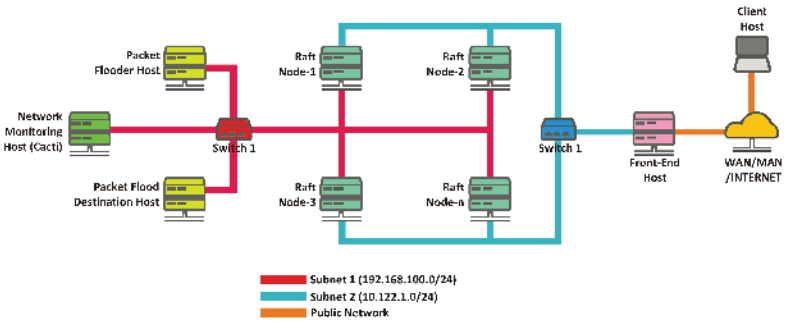
Dengan melihat data hasil pengujian pada Tabel 4.6 serta Tabel 4.7 maka dapat disimpulkan bahwa penggunaan besar *chunk* yang lebih besar mempunyai waktu *upload* yang lebih lama. Selain itu data dari percobaan kedua tidak mempunyai hasil *md5sum* yang yang sama.

Waktu *upload* yang lebih lama pada pengujian kedua dapat terjadi karena proses penyimpanan data dalam ukuran yang besar membutuhkan waktu yang lebih banyak dibandingkan penyimpanan data dengan data kecil. Hal ini berhubungan dengan kecepatan baca/tulis pada media penyimpanan komputer.

Data pengujian kedua tidak menghasilkan data yang valid karena data yang dikirimkan mempunyai ukuran yang terlalu besar untuk dikirimkan melalui *UDP*. Karena proses replikasi dilakukan dengan sangat cepat maka akan ada beberapa paket yang hilang jika paket datagram terlalu besar.

4.8 Pengujian Dengan Kondisi *Traffic Load* Jaringan yang Berbeda

Pada pengujian ini akan dilakukan *read and write* serta validasi data dengan tingkat beban jaringan yang berbeda. Kondisi beban jaringan dapat diubah dengan cara melakukan *packet flooding* pada sebuah jaringan untuk memberikan beban berat pada perangkat jaringan, dalam kasus penelitian ini akan memberikan beban pada perangkat *switch*. Topologi jaringan pengujian ini dapat dilihat pada Gambar 4.11.



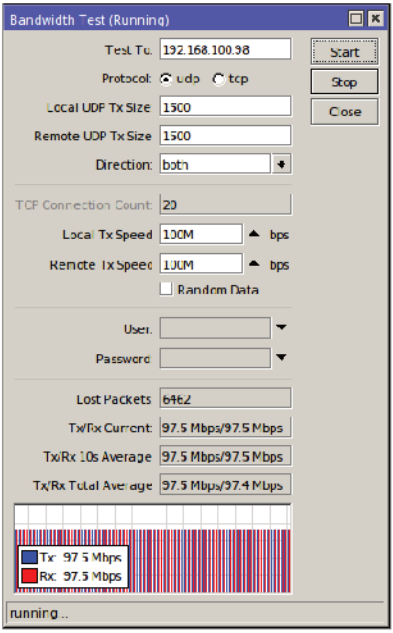
Gambar 4.11: Topologi pengujian dengan *flooding host* serta *monitoring host*

Sesuai dengan Gambar 4.11, terjadi penambahan *host* dimana dua *host* sebagai penerima serta pengirim *packet flood*, sedangkan satu *host* menjalankan *tools* monitoring yaitu *Cacti*. *Software* yang digunakan untuk melakukan *flooding* adalah *tools* dari *mikrotik* yaitu *bandwidth test*.

Pengujian replikasi dan validasi data akan dilakukan beberapa kali dengan mengubah besar *packet flood*. Paket yang digunakan dalam proses *flooding* adalah paket datagram. Besar paket yang akan dikirimkan yaitu sebesar *1Mbps, 5Mbps, 10Mbps, 50Mbps* dan *100Mbps*, dimana *100Mbps* adalah besar maksimal yang dapat ditangani oleh perangkat *switch*.

4.8.1 Proses pengujian

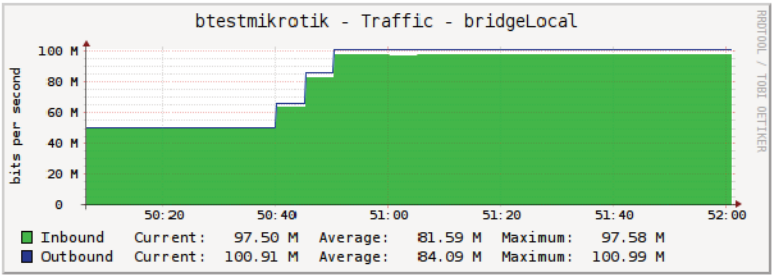
Langkah pertama pengujian adalah dengan melakukan pengiriman *udp packet flood*, proses ini dilakukan oleh *flooder host* dan dikirimkan ke *flood receiver*. Kedua *host* pada pengujian ini adalah perangkat *mikrotik board* dengan fitur *bandwidth test* yang telah diaktifkan. Proses ini dapat dilihat pada Gambar 4.12.



Gambar 4.12: Proses *packet flooding* sebesar *100Mbps*

Pada Gambar 4.12 alamat 192.168.100.98 adalah penerima *packet flood*, paket data yang dikirimkan adalah paket datagram karena menggunakan protokol *UDP*. Pada bagian bawah terdapat informasi jumlah paket yang *loss* dan juga terdapat grafik besar pengiriman serta penerimaan data secara *realtime*.

Langkah selanjutnya adalah memastikan bahwa perangkat *switch* sedang menangani beban trafik yang sama, untuk itu dapat dilihat melalui grafik pada *Cacti monitoring tool*.



Gambar 4.13: Grafik pada *Cacti* menunjukkan *traffic* yang sedang ditangani oleh perangkat *switch* secara *realtime*

Setelah memastikan beban yang sedang ditangani oleh *switch* maka proses *read/write* dan validasi data dapat dilakukan. Pengujian ini dilakukan dengan data yang sama dengan *load* jaringan yang berbeda. Data yang digunakan adalah data yang sama dengan data pengujian validasi. Hasil dari pengujian ini dapat dilihat pada Tabel 4.8. Dari data tabel tersebut dapat dilihat bahwa data tetap valid, namun di sisi lain ada variabel yang berpengaruh yaitu waktu. Data dari pengaruh *network traffic load* terhadap waktu dapat dilihat pada Tabel 4.9.

Tabel 4.8: Hasil pengujian validasi data dengan *Network Traffic Load* yang berbeda

No	Ukuran File	Traffic Load				
		1 Mbps	5 Mbps	10 Mbps	50 Mbps	100 Mbps
1	10 MB	Data Valid	Data Valid	Data Valid	Data Valid	Data Valid
2	50 MB	Data Valid	Data Valid	Data Valid	Data Valid	Data Valid
3	100 MB	Data Valid	Data Valid	Data Valid	Data Valid	Data Valid
4	500 MB	Data Valid	Data Valid	Data Valid	Data Valid	Data Valid
5	1 GB	Data Valid	Data Valid	Data Valid	Data Valid	Data Valid

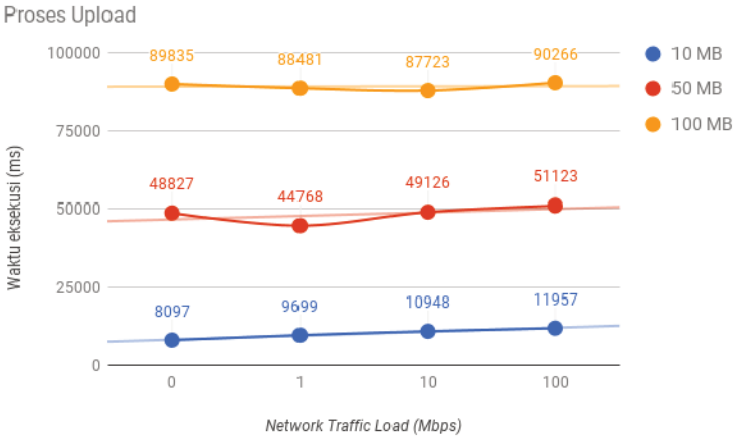
Tabel 4.9: Data hasil pengujian penggantian *Network Traffic Load* terhadap waktu penyimpanan data

No	Ukuran File	Traffic Load			
		Load Normal	1 Mbps	10 Mbps	100 Mbps
1	10 MB	8097 ms	9699 ms	10948 ms	11957 ms
2	50 MB	48827 ms	44768 ms	49126 ms	51123 ms
3	100 MB	89835 ms	88481 ms	87723 ms	90266 ms

Dengan melihat data pada Tabel 4.8, 4.9, serta 4.10, maka dapat disimpulkan bahwa sistem replikasi ini dapat berjalan dengan baik walaupun dengan *load* jaringan yang penuh. Untuk lebih gambaran yang lebih jelas dapat dilihat pada Gambar 4.14 serta 4.15, pada grafik tersebut dapat dilihat pada trafik jaringan yang tinggi maka waktu eksekusi semakin tinggi.

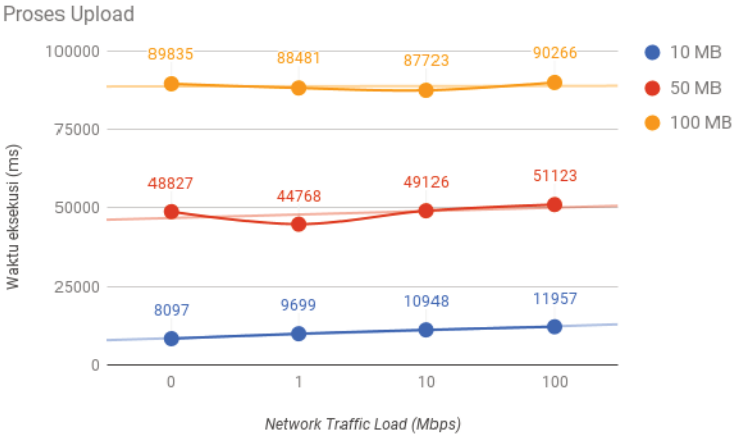
Tabel 4.10: Data hasil pengujian penggantian *Network Traffic Load* terhadap waktu pengambilan data

No	Ukuran File	Traffic Load			
		Load Normal	1 Mbps	10 Mbps	100 Mbps
1	10 MB	19,778 s	19,799 s	19,795 s	58,664 s
2	50 MB	98,990 s	99,891 s	100,126 s	295,320 s
3	100 MB	199,780 s	198,981 s	212,733 s	622,640 s



Gambar 4.14: Grafik proses *upload* data dengan perbedaan *network traffic load* terhadap waktu

Pada dasarnya paket data *UDP* sangat rentan terhadap *packet loss* pada jaringan yang penuh, namun dengan adanya Algoritma Konsensus Raft sistem ini dapat melakukan pengiriman kembali. Berbeda dengan aplikasi yang menggunakan *TCP* dimana proses *retransmission* terjadi pada *Transport Protocol*, sistem ini melakukan *checking* dan *retransmission* pada *layer* aplikasi.



Gambar 4.15: Grafik proses *download* data dengan perbedaan *network traffic load* terhadap waktu

Walaupun sistem ini dapat menyimpan data dengan valid, namun jika jaringan sedang menangani aliran trafik data yang tinggi maka dapat berpengaruh pada waktu penyimpanan dan waktu pengambilan data. Hal ini disebabkan karena terdapat banyak paket data yang *loss* selama proses replikasi maupun pengambilan data. Untuk mengatasi paket *loss* tersebut, sistem akan melakukan proses *retransmission* dengan berulang-ulang yang akan berdampak pada waktu eksekusi.

Halaman ini sengaja dikosongkan

BAB 5
PENUTUP

5.1 Kesimpulan

Dari hasil implementasi dan pengujian replikasi data menggunakan UDP serta Algoritma Konsensus Raft, maka dapat ditarik beberapa kesimpulan sebagai berikut :

1. *UDP* tetap dapat digunakan sebagai protokol replikasi yang *reliable*. Proses replikasi menggunakan *UDP* dilakukan dengan cara menulis *log index* pada setiap data yang akan dikirimkan sesuai dengan algoritma konsensus raft.

2. Kegagalan dalam pengiriman data menggunakan *UDP* dapat disebabkan oleh pengiriman paket yang terlalu besar. Oleh karena itu replikasi dapat dilakukan dengan melakukan pemecahan file *binary* menjadi *chunk data* yang kecil sehingga sistem replikasi ini sebenarnya adalah replikasi dari *chunk* data.

3. Karena replikasi data dilakukan dengan *chunk* data yang berurutan, maka proses pengambilan data berukuran besar dapat dilakukan sekaligus pada saat penyimpanan data sedang berjalan selama *key* telah diperoleh. Hal tersebut dapat dilakukan karena proses replikasi dilakukan secara berurutan dari *chunk* pertama hingga terakhir.

4. Sistem replikasi dengan Algoritma Konsensus Raft dan UDP dapat menjadi alternatif sistem replikasi yang *reliable*.

5.2 Saran

Demi pengembangan lebih lanjut mengenai tugas akhir ini, disarankan beberapa langkah lanjutan sebagai berikut :

1. Melakukan uji performansi serta uji keamanan data. Uji performansi sangat penting dilakukan untuk mengetahui perbandingan sistem replikasi ini dengan sistem replikasi lain.

2. Melakukan pengembangan pada protokol *UDP* yang digunakan dengan cara melakukan implementasi *ACK* pada setiap pengiriman paket datagram

3. Pengembangan penelitian dengan cara menggunakan jaringan yang lebih luas dan mempunyai alamat jaringan yang berbeda

4. Melakukan proses *tuning* dengan melakukan alalisis *metrics* antara kecepatan *upload/download*, beban *CPU*, *disk I/O*, serta besar *chunk* pengiriman data

5. Mengembangkan peneletian dengan melakukan implementasi menggunakan sistem klaster

DAFTAR PUSTAKA

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," Stanford University, 2014. (Dikutip pada halaman i, ix, 5, 6, 7, 8).

[2] M. T. Jones, "Boost application performance using asynchronous i/o," https://www.ibm.com/developerworks/library/l-async/, 2006. Terakhir diakses pada 16 April 2017. (Dikutip pada halaman ix, 9, 10).

[3] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," in Journal of Management Information Systems, Vol. 12, No. 4 (Spring, 1996), pp. 5–33, M.E. Sharpe, Inc., 2013. (Dikutip pada halaman 1).

[4] R. Catell, "Scalable sql and nosql data stores," http://www.catell.net/datastores/Datastores.pdf/, 2010. Terakhir diakses pada 2 Januari 2017. (Dikutip pada halaman 1).

[5] M. C. MAZILU, "Database replication," in Database Systems Journal vol. I, no. 2/2010, Academy of Economic Studies, Bucharest, Romania., 2010. (Dikutip pada halaman 1).

[6] S. R. Santosh Kumar, "Survey on transport layer protocols: Tcp udp," International Journal of Computer Applications (0975 - 8887) Volume 46- No. 7, India., 2012. (Dikutip pada halaman 1, 2, 8).

[7] W. Zhao, "Fast paxos made easy: Theory and implementation," Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH, IGI Global, USA, 2015. (Dikutip pada halaman 1, 5).

[8] "Consensus (computer science)," https://en.wikipedia.org/wiki/Consensus_(computer_science). Terakhir diakses pada 16 April 2017. (Dikutip pada halaman 5).

[9] H. Howard, "Arc: Analysis of raft consensus," in UCAML-CL-TR-857; ISSN 1476-2986, Cambridge University, United Kingdom, 2014. (Dikutip pada halaman 6, 7).

[10] "Fast portable non-blocking network programming with libevent," http://wangafu.net/~nickm/libevent-book. Terakhir diakses pada 16 April 2017. (Dikutip pada halaman 13).

BIOGRAFI PENULIS

Arif Nur Khoirudin, lahir pada 23 April 1995 di Bantul, DI Yogyakarta. Penulis lulus dari SMP Negeri 1 Piyungan pada tahun 2010 kemudian melanjutkan pendidikan ke SMA N 8 Yogyakarta hingga akhirnya lulus pada tahun 2013. Penulis kemudian melanjutkan pendidikan strata satu ke Jurusan Teknik Multimedia dan Jaringan ITS Surabaya yang kemudian berubah menjadi Teknik Komputer. Saat di kuliah penulis sempat mengikuti beberapa kegiatan antara lain robotika serta menjadi asisten aktif laboratorium B201.

Tak hanya aktif di lingkungan akademik saja, penulis sempat mengikuti beberapa organisasi. Pada tahun 2012 penulis sempat terpilih menjadi ketua karang taruna GMAG di Yogyakarta, selain itu hingga saat ini penulis sedang dipercaya sebagai Creative Director di sebuah organisasi peduli pendidikan iterasi keuangan anak muda yaitu Youth Finance Indonesia.

Penulis mengawali karir dengan menjadi pekerja paruh waktu sebagai system administrator dan junior backend programmer di sebuah perusahaan pengembang aplikasi pesan instan yaitu Catfiz Messenger. Pada saat ini penulis dipercaya sebagai CTO di Deftwork.com

Halaman ini sengaja dikosongkan